WRITING STAGING SQL STATEMENTS

The basic E.piphany extraction process consists of the following phases:

1. Loading data from source systems into the E.piphany staging tables
2. Running semantic instances against these staging tables.

This appendix describes the recommended practices for writing SQL statements that populate the dimension and fact staging tables.

BASE DIMENSION STAGING SQL STATEMENTS

Base dimensions describe the physical dimension tables in EpiMart. You can use EpiCenter Manager to configure the dimension columns for each base dimension table. During an extraction job, one or more SQL statements can be executed against the base dimension staging table to populate these columns.

The purpose of the base dimension staging query is to extract the latest values from the source system. You need not be concerned with determining when a value has changed because E.piphany's semantic templates perform this task.

You can use the Template feature in the SQL Statement dialog box to provide an SQL template for a given dimension table. For example, assume you define a base dimension called Customer with the following dimension columns:

- FullName
- Age
- City
- State
- ZipCode

In the SQL Statement dialog box, select the option **Populates dimension table.** Choose the dimension table (Customer) from the drop-down list. Clicking the Template button will display the following SQL in the dialog box:

```
SELECT
   <YOUR EXPRESSION>                       customer_sskey,
   $$TO_EPIDATE[ $$DBNOW ]                 date_modified,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]FullName,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]Age,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]City,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]State,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]ZipCode
FROM
   <YOUR TABLE>
```

Note: *The displayed template code also has a comment for each column that indicates the physical type of that column.*

This is a regular SQL statement for which you must provide the values <YOUR EXPRESSION> and <YOUR TABLE>. You must assign each column in the SELECT list a valid SQL expression for the value of its destination column. A FROM clause is required to make this a valid statement; a WHERE clause is optional.

An important point about these expressions is that null values are *not* allowed in any field of the staging table. The reason for this is that the E.piphany system uses GROUP BY statements at end-user query time to form the tables and charts of front-end applications. However, fact rows that aggregate on columns with null values are left out of the resulting reports because nulls are removed from GROUP BYs. Rather than incurring the query-time penalty for this check, EpiMart insists on non-null dimension column values.

To circumvent this problem, substitute the string 'UNKNOWN' for any null values using the $$NVL macro, as shown in the template. The E.piphany system will automatically generate an 'UNKNOWN' row in your dimension table. The 'UNKNOWN' value is configurable; if 'UNKNOWN' is a valid value in your dimension table, use another value.

The first two columns, **customer_sskey** and **date_modified**, were not specified in the definition of the Customer base dimension. These columns are implicitly added to the base dimension table by E.piphany's Adaptive Schema Generator and must be populated by the staging SQL. The first column is a source system key (**sskey**) and should be unique for every row in the staging table. The concept of **sskey** is important in EpiMart because it tells the semantic templates whenever a row in the staging table represents the same source system entity as a row that already exists in the dimension table. The **sskey** is a variable length string, normally of maximum length 50. It corresponds to the primary key of the source system table or the tables that make up this query.

Note: *If the **sskey** column is of interest to end users for querying, then a dimension column populated with the same values will need to be created because the **sskey** is not available for Web Page configuration.*

The other implicit column, **date_modified**, has the same name in all base dimension staging tables and is used to identify when a base dimension row is inserted into the EpiMart. If the source system contains a *creation date* field, then this field should be used. Otherwise, you can use the source system's expression for "right now," which causes newly extracted rows to assume the date when they were extracted into EpiMart; for Microsoft SQL Server this expression is `GetDate()`. The `$$DBNOW` macro is automatically expanded to the appropriate expression for the current date. For best results, dates should be returned as strings, for example, `5/26/1999`. This can be done using the `$$TO_EPIDATE` macro, as shown in the template.

The remainder of the columns in the `SELECT` list must be populated with an appropriate expression for the meaning of that column. Any SQL expression that can be executed against the source RDBMS is valid. Also, E.piphany provides a set of SQL macros that will be automatically expanded to the correct syntax for your source system. Use of macros facilitates the cross-platform usage of your SQL statements. See Appendix A, "E.piphany Macros," for more information.

In dimension tables, source system entries with large numbers of possible values are often classified into categories or ranges. For example, a source system may record the exact ages of individuals, while a dimension might simply record the age range into which an individual falls. This kind of categorization is called *binning*, and E.piphany's `$$CBIN_*` and `$$NBIN_*` macros are designed to simplify the process.

You can also use the drop-down menu next to the template button to insert a template for a single dimension column.

DUPLICATE SSKEYS

If during a single extraction, a staging table is loaded with two or more rows with the same **sskey**, then the last row entered is used. See Appendix F, "Semantic Types," for a description of the dimension semantic types.

DIMENSION STAGING QUERIES WITH JOINS

The E.piphany system allows the use of joins in base dimension staging queries. Star schemas typically de-normalize data structures in transactional systems into flat hierarchies, and you must be aware of what the granularity of a base dimension represents in this circumstance.

For example, you will rarely want to use a Cartesian product of two tables in a base dimension staging query, unless the **sskey** of the result set will combine the primary keys of the two tables that are being crossed. It is more common for a single table to "drive" the result set, with other tables joined through unique key lookups to provide additional textual values. For instance, a Product Master table in the source system might represent the driving table of a Product base dimension (with the **sskey** taken from the primary key of the Product Master table), but other tables with textual values for Product Line or Platform may be joined with this master table. In this case, you should ensure that the joined columns of the lookup tables are properly indexed (usually with UNIQUE indexes).

BASE DIMENSION QUERIES WITH DISTINCT FACT VALUES

Sometimes dimensions are created for which no corresponding master table exists in the source system. For instance, an Order fact may have an Order type associated with it (with several possible choices). These values are embedded directly in the rows of the source system's Order table, but no lookup table exists with all the choices. In this case, a `SELECT DISTINCT` query against the Order type column of the source system's Order table might be appropriate for populating base dimension staging tables in EpiMart. The alternative to this method is the use of degenerate dimensions in the fact table, although degenerate dimensions cannot be aggregated.

SEED DIMENSION EXTRACTION

If you are using seed dimensions in campaigns, you also need to populate the **IndivSeed** and **GroupSeed** dimensions. These dimensions have the same columns as the dimensions that correspond to the **indiv** and **group** dimension roles, respectively. You should ensure that your seed dimensions have more rows than the maximum number of cells that a user will have in a campaign, since each cell will be assigned a distinct element of the seed dimension.

Campaign and Cell Extraction

If you are using the Campaign manager, you need to extract data for the **Campaign** and **Cell** dimensions from the backfeed tables. E.piphany provides default extractors for this purpose, but you will need to modify these extractors if you have added additional columns to the **Campaign** or **Cell** dimensions. The default extractor for the Campaign dimension has the following SQL code:

```
SELECT
    $$NVL[ campaign_id ~,~ 'UNKNOWN' ]        campaign_sskey,
    $$TO_EPIDATE[ rundatetime ]               date_modified,
    $$NVL[ campaign_name ~,~ 'UNKNOWN' ]      campaign_name,
    $$NVL[ campaign_label ~,~ 'UNKNOWN' ]     campaign_label,
    $$NVL[ fixed_setup_cost ~,~ ' ' ]         fixed_setup_cost,
    $$NVL[ def_rev_per_response ~,~ ' ' ]     def_rev_per_response,
    $$NVL[ def_profit_per_response~,~' ']    def_profit_per_response,
    $$NVL[ def_cost_per_treatment ~,~ ' ' ] def_cost_per_treatment,
    $$NVL[ def_response_rate ~,~ ' ' ]        def_response_rate,
    $$NVL[ campaign_id ~,~ 'UNKNOWN' ]        campaign_id,
    $$NVL[ campaign_code ~,~ 'UNKNOWN' ]      campaign_code,
FROM
    Campaign_BF$$CURR
```

Ordinarily, these extractors will simply copy values from backfeed-table columns to the dimension columns with the same names. As usual, the `$$NVL` macro should be used to ensure that no null values are copied. The backfeed table has a column called rundatetime, which records the time that the campaign was run, that should be used for the date. The value of **campaign_id** is generally also used for the source system key. You may want to use the `$$NBIN_*` and `$$CBIN_*` macros to divide numerical values (such as costs) into ranges. The backfeed tables for the Campaign table are named **Campaign_BF_A** and **Campaign_BF_B**, so data should be extracted from `Campaign_BF$$CURR`.

The same considerations apply to the extractor for the Cell table.

The built-in columns of the Campaign and Cell dimensions are described on page 443.

Note: *While Campaign and Cell dimension data is ordinarily extracted from the backfeed tables, these dimensions can also be populated from external data sources.*

FACT STAGING SQL STATEMENTS

SQL statements that populate fact staging tables are generally more complex than the ones used to load dimension staging tables. As with base dimension tables, the columns of the SELECT statements are determined by the metadata definition of the fact table in addition to certain implicit rules.

To illustrate this point, assume that you define an Order fact with the following dimension roles:

- CustomerBillTo
- Product
- CustomerShipTo
- SalesPerson

The EpiCenter contains a single degenerate dimension called OrderNumber, and the Order table has two fact columns: **net_price** and **number_units** that represent the extended amount for an order line item, along with the quantity.

Clicking the Template button on the SQL Statement dialog box (with the **Populates fact table** option selected and the Order table selected in the drop-down list) displays the following SQL in the dialog box:

```
SELECT
   <YOUR EXPRESSION>                       ss_key,
   $$TO_EPIDATE[ <YOUR EXPRESSION> ]       date_key,
   <YOUR EXPRESSION>                       transtype_key,
   <YOUR EXPRESSION>                       process_key,
   $$NVL[ <YOUR EXPRESSION> -,- 'UNKNOWN' ]customerbillto_sskey,
   $$NVL[ <YOUR EXPRESSION> -,- 'UNKNOWN' ]product_sskey,
   $$NVL[ <YOUR EXPRESSION> -,- 'UNKNOWN' ]customershipto_sskey,
   $$NVL[ <YOUR EXPRESSION> -,- 'UNKNOWN' ]salesperson_sskey,
   $$NVL[ <YOUR EXPRESSION> -,- 'UNKNOWN' ]ordernumber_key,
   $$NVL[ <YOUR EXPRESSION> -,- 0 ]        net_price,
   $$NVL[ <YOUR EXPRESSION> -,- 0 ]        number_units
FROM
   <YOUR TABLE>
```

As with base dimension staging queries, you must identify what a row in this fact table represents. Based on the columns in this example, a row seems to indicate a line item of a sales order. (In this case, the assumption is that the salesperson gets full credit for a line item; another interpretation of this fact row might be a particular amount of credit that a salesperson received for an order line item.) Typically, the FROM clause of this query would join the Order Line Item table to the Order Header table in the source system.

The columns in the SELECT list can now be divided into these categories:

- Implicit columns that were added automatically
- Dimension role foreign keys
- Degenerate dimension keys
- Fact numeric columns

First, consider implicit columns. As with base dimensions, each fact staging row contains an **ss_key** (notice the difference in spelling) that uniquely identifies this row in the source system. In this example, the **ss_key** might be a concatenation of the Order Number with the Order Line Number (since this combination is presumably unique). **ss_key**s will be used on subsequent extractions to prevent duplicate copies of the fact row from being created in EpiMart.

The **date_key** indicates when the fact occurred. Since time is a central component of EpiMart, each fact table must contain this column. Many facts are time based; in this example, **date_key** represents the time when the order was placed. However, if time is not important for this fact, then the current system time can be used as a placeholder. Note that **date_key** is granular only to that single *day* when the fact occurred. For best results, the fact SQL Statement should return the day as a string, for instance, 5/1/1999. E.piphany recommends that you use the `$$TO_EPIDATE` macro to ensure that dates are in the proper format.

Transaction type is another central concept of fact table processing in EpiMart. The SQL statement should return a numeric key that matches with one of the transaction types defined on the Configuration dialog box in EpiCenter Manager. For example, the Order fact might hold both Bookings and Shippings, and **transtype_key** would identify which fact staging rows were which. (See Appendix F, "Semantic Types," for more information about **transtype_key**.)

Note: *Transaction type values in the range 10000-20000 are reserved for use by E.piphany.*

A fact staging table can contain different types of rows that need to be handled in different ways by the semantics. The **process_key** identifies rows from the fact table to be processed in a specific way by a semantic instance. A value of 1 indicates a transactional fact, and a value of 2 indicates a state-like fact.

Next, you must enter values for each of the dimension role foreign keys. Notice that the names of the columns in the SQL template are **DimRoleName_sskey**. These fields refer back to **sskeys** of the base dimension tables for this fact. You need to understand the meaning of each base dimension table to ensure that the keys resolve properly. If the **sskey** of the Product base dimension is taken from a Product Master list in the source system, then **product_sskey** in the Order table must also refer to an entry in the Product Master. If a base dimension is the cross-product of two source system tables, the fact staging keys for that dimension must also represent a unique cross-product entry.

Note: *If you have defined additional dimension roles that refer to the date dimension, then the names of the foreign keys for those roles end in* key *rather than* sskey. *For example, if you have defined a dimension role called* ***inquiry_date*** *that refers to the date base dimension, then the name of the foreign key for that dimension role is* ***inquiry_date_key***.

Degenerate keys in the fact staging query should be populated with string values. In the example above, the **ordernumber_key** field would probably be populated with the actual primary key of the Order Header table, such as Order Number 253AD56.

Finally, the numeric columns represent the actual quantities and raw amounts that are associated with each fact entry. Each column should be an additive amount for correct front-end query results. For instance, total dollar amounts for a line item should be populated instead of unit prices because unit prices cannot be added across fact rows.

IND_GROUP_JOINER EXTRACTION

If you are using the List Manager or Campaign Manager, you need to extract data for the **Ind_Group_Joiner** fact table. This table specifies the association between members of the dimensions that have the **indiv** and **group** dimension roles. The usual format for **Ind_Group_Joiner** extraction is the following:

```
SELECT
   <YOUR EXPRESSION>                       ss_key,
   $$TO_EPIDATE[ $$DBNOW ]                 date_key,
   1                                       transtype_key,
   2                                       process_key,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]group_sskey,
   $$NVL[ <YOUR EXPRESSION> ~,~ 'UNKNOWN' ]indiv_sskey,
   $$NVL[ <YOUR EXPRESSION> ~,~ 0 ]'       member
FROM
   <YOUR TABLE>
```

Each member of the dimension with the indiv role must be associated with exactly one member of the dimension with the group role. Note the following:

- The **member** column indicates that the individual with **indiv_sskey** belongs to the group with **group_sskey**. The **member** column can have any non-zero value. In most cases, the value 1 is used.
- The value of **process_key** should be 2. Note that this implies that new data should be merged into the **Ind_group_joiner** fact table using a Transactional/State-Like semantic type (see "Transactional/State-like," on page 455).
- Generally, the value of **ss_key** should be the same as the value of **indiv_sskey**. Making these values the same ensures that the Transactional/State-Like semantic type properly updates the table when an individual changes group membership.
- Ordinarily, the only **transtype** value that is used in the **Ind_group_joiner** table is 1.

SEEDJOINER EXTRACTION

If you are using seed dimensions in campaigns, you need to populate the **SeedJoiner** fact table. The **SeedJoiner** fact table has the same role for the **IndivSeed** and **GroupSeed** dimensions that the **Ind_Group_Joiner** has for the individual and group dimensions. Therefore, extraction for the **SeedJoiner** is completely analogous to extraction for **Ind_Group_Joiner**. A row of the **SeedJoiner** table specifies that the member of **IndivSeed** with **sskey** value **indiv_sskey** belongs to the member of **GroupSeed** with **sskey** value **group_sskey**.

Note that only seed dimension members that have SeedJoiner fact table entries are available for seeding. Therefore, you should ensure that you have a SeedJoiner fact for every member of your seed dimensions.

COMMUNICATION EXTRACTION

If you are using the Campaign manager, you also need to extract data for the **Communication** fact table from the backfeed table. E.piphany provides a default extractor for this purpose. The default extractor for the Communication table has the following SQL code:

```
SELECT
   ($$TO_YYYYMMDD[rundatetime] $$CAT $$TO_HHMMSS[rundatetime] $$CAT
      group_sskey $$CAT indiv_sskey $$CAT cell_sskey) ss_key,,
   $$TO_EPIDATE[ rundatetime ]             date_key,
   $$TRANSTYPE[BACKFEED]                   transtype_key,
   1                                       process_key,
   group_sskey,
   indiv_sskey,
   campaign_sskey,
   cell_sskey,
   treatment_code_key,
   1                                       occur
FROM
   Communication_BF$$CURR
```

As with the extractors for the Campaign and Cell dimensions, most columns are simply copied from the backfeed table. Note the following:

- The communication backfeed table contains a column called **rundatetime** that records the time at which the campaign was run.
- The default **transtype** value for a communication, which indicates that a treatment has been applied, is entered using the `$$TRANSTYPE[BACKFEED]` macro. Additional values may be defined for other campaign-related facts.
- The **occur** column indicates that the fact has occurred. It should be given a value of `1`.

USING EXTERNAL TABLES AS INPUTS TO STAGING QUERIES

Sometimes it may be necessary to bring data into EpiMart external (temporary) tables before performing any joins; for example, if the source system's SQL limits your ability to manipulate the data. The full power of EpiMart's RDBMS engine can then be used to load the staging tables. In this case, the following sequence of actions is usually employed:

1. Drop any indexes on the external tables for fast loading.
2. Load the external tables from the source system.
3. Create any indexes on external tables needed for fast joins.
4. Load the staging tables using queries against the EpiMart External tables. All query plans should use the indexes built in Step 3.

BUILT-IN COLUMNS IN CAMPAIGN AND CELL DIMENSIONS

The Campaign and Cell dimensions have several built-in columns that cannot be removed or modified. These columns ar found in both the dimension tables and the backfeed tables.

The Campaign dimension has the following built-in columns:

- **campaign_code**: A unique identifying code for the campaign

  In the backfeed table, this code generally has a value that has been assigned by the end user. If the end user does not assign this code when saving a campaign, then a unique string is automatically generated the first time the campaign is saved. This is the backfeed table field that is used to identify a campaign for exclusion purposes.
- **campaign_id**: An identifying code for the campaign version

  In the backfeed table, this value is automatically incremented every time a campaign is saved.
- **campaign_label**: A descriptive identifier for the campaign
- **def_cost_per_treatment**: The default cost per treatment in the campaign
- **def_profit_per_response**: The default profit per response in the campaign
- **def_response_rate**: The default response rate for the campaign
- **def_rev_per_response**: The default revenue per response in the campaign
- **fixed_setup_cost**: The fixed setup cost of the campaign

The Cell dimension has the following built-in columns:

- **treatment_code**: A unique identifying code for the cell

  In the backfeed table, this code generally has a value that has been assigned by the end user. If the end user does not assign this code when saving a campaign, then the value is set to `'N/A'`. This is the backfeed table field that is used to identify a cell for exclusion purposes.
- **campaign_id**: The version code for the associated campaign
- **cell_id**: A unique identifier for the cell and campaign version

  In the backfeed table, this is formed by concatenating **campaign_id** with the cell node number. Within a campaign, this code is unique.
- **cell_label**: A descriptive identifier for a cell
- **cell_position**: The position of the cell in the segmentation tree
- **cell_size**: The number of individuals or groups in the cell

- **est_cell_cost**: The estimated cost of the cell
- **est_cell_profit**: The estimated profit for the cell
- **est_cell_response_rate**: The estimated response rate for the cell

  In the backfeed table, if this value has not been specified by the end user, then the **def_response_rate** value of the associated campaign is used.
- **est_cell_revenue**: The estimated revenue rate for the cell
- **est_profit_per_response**: The estimated profit per response for the cell

  In the backfeed table, if this value has not been specified by the end user, then the **def_profit_per_response** value of the associated campaign is used.
- **est_revenue_per_response**: The estimated revenue per response for the cell

  In the backfeed table, if this value has not been specified by the end user, then the **def_rev_per_response** value of the associated campaign is used.
- **output_channel_label**: The label of the output channel associated with the cell
- **output_format_label**: The label of the output format associated with the cell
- **output_processor_label**: The label of the output processor associated with the cell
- **treatment_unit_cost**: The cost of a single treatment in the cell

  In the backfeed table, if this value has not been specified by the end user, then the **def_cost_per_treatment** value of the associated campaign is used.